Inteligencia Artificial 2017-2018

Práctica 1: LISP

Celia San Gregorio Moreno
Álvaro Martínez Morales

Grupo: 2213
Fecha: 03/03/2018

Índice

1. Similitud Coseno

Apartado 1.1

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prod-escalar-rec (x y)
;;;
;;; Calcula el producto escalar de dos vectores de forma recursiva.
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;              y: vector, representado como una lista
;;; OUTPUT: producto escalar de x e y
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun prod-escalar-rec (x y)
  (if (or (null x) (null y))
     0
    (+ (* (first x) (first y)) (prod-escalar-rec (rest x) (rest y)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-rec (x y)
;;;
;;; Calcula la similitud coseno de un vector de forma recursiva
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;; La semejanza coseno entre dos vectores que son listas vacías o que son
;;; (0 0...0) es NIL.
;;;
;;; INPUT: x: vector, representado como una lista
;;;              y: vector, representado como una lista
;;; OUTPUT: similitud coseno entre x e y
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun sc-rec (x y)
  (if (or (= (prod-escalar-rec x x) 0) (= (prod-escalar-rec y y) 0))
     nil
    (/ (prod-escalar-rec x y) (* (sqrt (prod-escalar-rec x x)) (sqrt (prod-escalar-rec y y))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prod-escalar-mapcar (x y)
;;;
;;; Calcula el producto escalar de dos vectores usando mapcar.
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;              y: vector, representado como una lista
;;; OUTPUT: producto escalar de x e y
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun prod-escalar-mapcar (x y)
  (if (or (null x) (null y))
     nil
    (reduce #'+ (mapcar #'* x y))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-mapcar (x y)
;;;
;;; Calcula la similitud coseno de un vector usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;; La semejanza coseno entre dos vectores que son listas vacías o que son
```

```
;;; (0 0...0) es NIL.
;;;
;;; INPUT: x: vector, representado como una lista
;;;              y: vector, representado como una lista
;;; OUTPUT: similitud coseno entre x e y
(defun sc-mapcar (x y)
  (if (or (= (prod-escalar-mapcar x x) 0) (= (prod-escalar-mapcar y y) 0))
     nil
   (/ (prod-escalar-mapcar x y) (* (sqrt (prod-escalar-mapcar x x)) (sqrt (prod-escalar-mapcar y y))))))
```

Similitud coseno de forma recursiva

Para implementar la similitud coseno entre dos vectores de forma recursiva, primero definimos una función auxiliar `prod-escalar-rec (x y).`

Esta función comprueba primero si x o y son NIL; si lo son devolverá el valor 0, ya que si uno de los vectores es una lista vacía, ello supone multiplicar por 0 cada uno de los elementos del otro vector.

En caso de que ni x ni y sean NIL, calcula la suma de los productos de los primeros elementos de x e y, y de los productos del resto de los elementos. La función termina cuando llega al final de los vectores.

En cuanto a `sc-rec(x y)`, primero comprueba si los módulos de los vectores (o el producto escalar de sí mismos) es 0. Si lo es, devuelve 0 para evitar un caso de división por 0.

Si el módulo de x e y es distinto de 0, calcula la similitud coseno siguiendo la fórmula:

$$\cos_{similarity}(x, y) = \frac{\sum_{i=1}^{n} xiyi}{\sqrt{\sum_{i=1}^{n} xi^2}\sqrt{\sum_{i=1}^{n} yi^2}}$$

Similitud coseno usando mapcar

Para implementar la similitud coseno entre dos vectores usando mapcar, también definimos una función auxiliar `prod-escalar-mapcar (x y).`

Esta función realiza el producto sobre cada uno de los elementos de x e y utilizando mapcar, y suma cada producto mediante una llamada a reduce. De igual forma que en `prod-escalar-rec (x y)`, si ambos vectores son NIL devuelve 0.

Por último, `sc-mapcar(x y)` realiza el mismo cálculo que `sc-rec(x y)`,, pero con llamadas a `prod-escalar-rec (x y)` para realizar las operaciones. De nuevo, si los módulos de x o y son 0, devolverá 0.

Apartado 1.2

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; crea-vectores-conf (cat vs conf)
;;;
```

```
;;; Devuelve un lista de vectores cuya similitud respecto
;;; a una categoria es mayor que conf (nivel de confianza)
;;;
;;; INPUT: cat: vector que representa a una categoría, representado como una lista
;;;              vs: vector de vectores
;;;              conf: Nivel de confianza
;;; OUTPUT: Vectores cuya similitud con respecto a la categoría es superior al
;;; nivel de confianza. No están ordenados.
(defun crea-vectores-conf (cat vs conf)
  (if (null vs)
      nil
    ;Obtiene la similitud coseno entre
    ;el vector categoría y el primer vector de vs.
    (let ((similitud-cos (sc-rec cat (first vs))))
      ;Comprueba si es distinta de NIL y es mayor
      ;que el parámetro conf.
      (if (and (not (null similitud-cos))
               (> similitud-cos conf))
          ;Si cumple las condiciones, crea una lista con el vector
          ;y el resto de los elementos de vs.
          (append (list (first vs)) (crea-vectores-conf cat (rest vs) conf))
        ;Sino, sigue evaluando vs.
        (crea-vectores-conf cat (rest vs) conf)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf (cat vs conf)
;;;
;;; Devuelve aquellos vectores similares a una categoria
;;;
;;; INPUT: cat: vector que representa a una categoría, representado como una lista
;;;              vs: vector de vectores
;;;              conf: Nivel de confianza
;;; OUTPUT: Vectores cuya similitud con respecto a la categoría es superior al
;;; nivel de confianza, ordenados
(defun sc-conf (cat vs conf)
  (sort (crea-vectores-conf cat vs conf) #'> :key (lambda (x) (sc-rec x cat))))
```

Para obtener un conjunto de vectores ordenado según su similitud a una categoría dada, `sc-conf (cat vs conf)` utiliza la función auxiliar `crea-vectores-conf (cat vs conf).`

Esta función auxilar, para cada vector del conjunto de vectores, comprueba si su similitud coseno con respecto a cat es distinta de NIL y es además mayor que el nivel de confianza conf.

Si la condición se cumple, añade el vector a una lista de vectores cuya similitud coseno es mayor que conf. En caso contrario, ignora el vector y evalúa el resto de vectores del conjunto de manera recursiva hasta que no queden más elementos.

En cuanto a `sc-conf (cat vs conf)`, la función obtiene la lista de vectores con similitud coseno superior a conf y la ordena de mayor a menor similitud.

Apartado 1.3

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; crea-vectores-similares (cats text func)
;;;
;;; Devuelve una lista de listas con elementos de tipo:
;;; ((<id-cat-1> <sc-1>) (<id-cat-2> <sc-2>) ... (<id-cat-n> <sc-n>))
;;;
;;; Siendo <id-cat> un identificador de categoria que pertenece a
;;; cats y <sc-n> el resultado de la similitud coseno entre el vector
;;; text y dicha categoría.
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;;              text: vector que representa un texto, representado como una lista
;;;              func: función para evaluar la similitud coseno
;;; OUTPUT: Pares identificador de categoría con resultado de
;;; similitud coseno, sin ordenar
(defun crea-vectores-similares (cats text func)
  ;Si hemos llegado al final del vector de categorías,
  ;no hay más cálculos que hacer.
  (if (null cats)
      nil
    ;Sino, obtiene el id de la categoría, el
    ;vector categoría actual y el vector texto actual.
    (let ((id_categoria (first (first cats)))
          (categoria (rest (first cats)))
          (texto (rest text)))
      ;Construye una lista con pares de tipo
      ;(<id-categoria> <similitud-coseno>).
      (cons (list id_categoria (funcall func categoria texto))
            (crea-vectores-similares (rest cats) text func)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; vector-mas-similar-rec (elem vectors)
;;;
;;; Devuelve el par (<id-categoria> <similitud-coseno>) con mayor
;;; similitud coseno de un vector de vectores con ese formato
;;; de elementos.
;;;
;;; El parámetro elem contiene el vector con mayor similitud
;;; coseno que se ha encontrado hasta el momento.
;;;
;;; INPUT: elem: par (<id-categoria> <similitud-coseno>), representado como una lista
;;;              vectors: vector de vectores, representado como una lista de listas
;;; OUTPUT: Par (<id-categoria> <similitud-coseno>) con mayor valor
;;;                                  de similitud coseno.
(defun vector-mas-similar-rec (elem vectors)
  ;Obtiene max-sc, la máxima similitud coseno encontrada
  ;hasta el momento (se usará como referencia); y
  ;similitud-cos, la <similitud-coseno> del primer
  ;elemento del vector de vectores.
  (let ((max-sc (second elem))
        (similitud-cos (second (first vectors))))
    ;Si sólo queda un elemento del vector de vectores
    ;por comparar, devolverá el par <id-categoria> <similitud-coseno>)
    ;con mayor similitud coseno.
    (if (null (rest vectors))
        (if (> similitud-cos max-sc)
            (first vectors)
          elem)
      ;Sino, seguirá recorriendo el vector de vectores con elem
```

```
    ;o un vector de la lista de listas dependiendo de cúal
    ;tenga mayor similitud coseno.
    (if (> similitud-cos max-sc)
       (vector-mas-similar-rec (first vectors) (rest vectors))
     (vector-mas-similar-rec elem (rest vectors))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-vector-mas-similar (cats text func)
;;;
;;; A partir de una lista de tipo:
;;; ((<id-cat-1> <sc-1>) (<id-cat-2> <sc-2>) ... (<id-cat-n> <sc-n>))
;;;
;;; Obtiene el par (<id-cat> <similitud-coseno>) con
;;; la similitud coseno más alta.
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;;              text: vector que representa un texto, representado como una lista
;;;              func: función para evaluar la similitud coseno
;;; OUTPUT: Par (<id-categoria> <similitud-coseno>) con mayor valor
;;;                                    de similitud coseno.
(defun vector-mas-similar (cats text func)
  (let ((vectors (crea-vectores-similares cats text func)))
    (vector-mas-similar-rec (first vectors) vectors)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-classifier (cats texts func)
;;;
;;; Clasifica a los textos en categorías.
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;;              texts: vector de vectores, representado como una lista de listas
;;;              func: función para evaluar la similitud coseno
;;; OUTPUT: Pares identificador de categoría con resultado de similitud coseno
;;;
(defun sc-classifier (cats texts func)
  ;Si hemos llegado al final del vector de textos,
  ;devolverá NIL (no hay más similitudes-coseno que evaluar).
  (if (null texts)
     nil
    ;Sino, crea una lista con los pares (<id-categoria> <similitud-coseno>)
    ;con la similitud coseno más alta entre cada texto y las distintas categorías.
    (cons (vector-mas-similar cats (first texts) func) (sc-classifier cats (rest texts) func))))
```

Para obtener una lista de pares (<id-categoría> <similitud-coseno>) a partir de un vector de vectores con categorías y otro con textos, la función `sc-classifier (cats texts func)` se sirve de varias funciones auxiliares:

1. **`crea-vectores-similares (cats text func).`** A partir de un vector de vectores que representa las distintas categorías, un vector que representa un texto y la función de similitud coseno, `crea-vectores-similares (cats text func)` va construyendo una lista de listas.

   Cada elemento de esa lista de listas es un par (<id-categoría> <similitud-coseno>) que se obtiene al calcular la similitud coseno entre el texto y cada una de las categorías contenidas en cats, todo ello de forma recursiva.
2. **`vector-mas-similar-rec (elem vectors).`** A partir de un vector de vectores de tipo ((<id-categoría> <sc-1>) (<id-categoría> <sc-2>) ...

(<id-categoría> <sc-n>)), devuelve el par (<id-categoría> <sc>) con mayor valor de similitud coseno.

Inicialmente, toma como referencia el primer elemento de vectors (elem); su similitud coseno será la máxima encontrada hasta el momento.

Si al recorrer cada vector de la lista la función encuentra un elemento con mayor similitud coseno que elem, lo tomará como referencia y evaluará el resto de la lista. En caso contrario, ignorará el elemento y seguirá manteniendo elem como referencia.

Todo el recorrido se realiza de forma recursiva.

3. **`vector-mas-similar (elem vectors).`** Obtiene la lista de pares (<id-categoría> <similitud-coseno>) y devuelve el que mayor similitud coseno tenga.

A partir de estas funciones, `sc-classifier (cats texts func)` construye de forma recursiva una lista de pares (<id-categoría> <similitud-coseno>) con el mayor valor de similitud coseno que existe entre cada texto y las categorías del vector de vectores cats.

Apartado 1.4

Código

```
;;
;; EJEMPLOS:
;;
(sc-classifier '((1 43 23 12) (2 33 54 24)) '((1 3 22 134) (2 43 26 58)) #'sc-rec)
;; ((2 0.48981872) (1 0.81555086))
(sc-classifier '((1 43 23 12) (2 33 54 24)) '((1 3 22 134) (2 43 26 58)) #'sc-mapcar)
;; ((2 0.48981872) (1 0.81555086))

(sc-classifier '((1 30 11 90 4) (2 200 3 25 88)) '((1 120 10 80 3) (2 31 20 85 55)) #'sc-rec)
;; ((2 0.82673454) (1 0.8757294))
(sc-classifier '((1 30 11 90 4) (2 200 3 25 88)) '((1 120 10 80 3) (2 31 20 85 55)) #'sc-mapcar)
;; ((2 0.82673454) (1 0.8757294))

(sc-classifier '((1 14 60) (2 50 75) (3 2 1)) '((1 90 4) (2 5 6) (3 45 60)) #'sc-rec)
;; ((3 0.91340166) (2 0.9943091) (2 0.9984604))
(sc-classifier '((1 14 60) (2 50 75) (3 2 1)) '((1 90 4) (2 5 6) (3 45 60)) #'sc-mapcar)
;; ((3 0.91340166) (2 0.9943091) (2 0.9984604))

(time (sc-classifier '((1 43 23 12) (2 33 54 24)) '((1 3 22 134) (2 43 26 58)) #'sc-rec))
(time (sc-classifier '((1 43 23 12) (2 33 54 24)) '((1 3 22 134) (2 43 26 58)) #'sc-mapcar))

(time (sc-classifier '((1 30 11 90 4) (2 200 3 25 88)) '((1 120 10 80 3) (2 31 20 85 55)) #'sc-rec))
(time (sc-classifier '((1 30 11 90 4) (2 200 3 25 88)) '((1 120 10 80 3) (2 31 20 85 55)) #'sc-mapcar))

(time (sc-classifier '((1 14 60) (2 50 75) (3 2 1)) '((1 90 4) (2 5 6) (3 45 60)) #'sc-rec))
(time (sc-classifier '((1 14 60) (2 50 75) (3 2 1)) '((1 90 4) (2 5 6) (3 45 60)) #'sc-mapcar))
```

Realizamos una serie de pruebas con vectores de textos y categorías de distintos tamaños, cuyos resultados están comentados en el código.

Además, medimos los tiempos de ejecución y observamos lo siguiente:

- En todas las llamadas a sc-classifier con #'sc-rec y #'sc-mapcar, la versión mapcar tarda significativamente más ciclos de reloj en completar la operación.

- Cuantos más elementos tenga cada lista de cats y texts, la función recursiva #'sc-rec y sc-classifier tardan menos ciclos de reloj en terminar la operación.

- Por el contrario, si un vector tiene menos elementos, sc-classifier tarda más ciclos de reloj en computar el resultado.

2. Raíces de una función

Apartado 2.1

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bisect (f a b tol)
;;;
;;; Calcula la raiz de una función dada en un intervalo dado mediante
;;; la técnica de la bisección a partir de un punto de tolerancia
;;; determinado.
;;;
;;; INPUT:		f: Función de la que calcular la raiz
;;;			a: Mínimo del intervalo
;;;				b: Máximo del intervalo
;;;				tol: Resultado mínimo para aceptar un resultado
;;; OUTPUT: Raiz de la función en el intervalo establecido
;;;
(defun bisect (f a b tol)
 ;Genera x en en ámbito de la ejecución actual de la función de bisectriz
 (let ((x (/ (+ a b) 2)))
  (if (>= (* (funcall f a) (funcall f b)) 0)
    ;Si f(a) y f(b) son ambas positivas o negativas, devuelve NIL
    nil
   (if (< (- b a) tol)
     ;Si b - a < tol, la función devuelve x como resultado
     x
    ;Si f(a) * f(x) < 0, la función busca en [a, x]
    (if (< (* (funcall f a) (funcall f x)) 0)
      ;Si no, reposicionamos uno de los puntos como x y continuamos.
      (bisect f a x tol)
     (bisect f x b tol)))))
```

Para encontrar una raíz en el intervalo [a, b], la función `bisect (f a b tol)` sigue varios pasos.

En primer lugar, calcula x como el punto medio entre a y b. Después, multiplica los resultados de aplicar el método para encontrar la raíz sobre a y b (es decir, f(a) * f(b)). Si detecta que el producto es mayor o igual que cero, devuelve NIL (ya que no es seguro que encuentre raíces).

En caso contrario, comprueba si b – a tiene un valor menor que la tolerancia mínima permitida. Si es menor, hemos encontrado la raíz x. Sino, la función multiplicará f(a) * f(x) para buscar la raíz en [a, x] o [x, b] de forma recursiva, dependiendo del signo de f(a) * f(x)

Apartado 2.2

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; alrrot (f lst tol)
```

```
;;;
;;; Calcula la raiz de una función dada en cada par de intervalos
;;; consecutivos de una lista ordenada
;;;
;;; INPUT:          f: Función de la que calcular la raiz
;;;                 lst: Listado de intervalos
;;;                         tol: Resultado mínimo para aceptar un resultado
;;; OUTPUT: Listado de las raices de la función en los intervalos establecidos
;;;
(defun allroot (f lst tol)
  (if (null (rest lst))
     ;Si no quedan al menos dos números en la lista, la recursividad termina.
     nil
   ;Comprueba si el signo de f(lst[i]) y f(lst[i+1]) es distinto.
   ;Es decir, que  f(lst[i]) * f(lst[i+1]) < 0
   (if (< (* (funcall f (first lst)) (funcall f (second lst))) 0)
      ;Calcula la bisectriz de los dos primeros elementos de la lista y pasa
      ;la lista a la llamada recursiva, quitando el primer elemento
      (cons (bisect f (first lst) (second lst) tol) (allroot f (rest lst) tol))
     ;Si no se cumple la condición, continúa evaluando los
     ;elementos de la lista.
     (allroot f (rest lst) tol))))
```

Para calcular la raíz en cada par de intervalos de una lista, `allroot (f lst tol)` realiza una serie de cálculos.

Primero, comprueba si hay al menos dos números en la lista que pueda tomar como intervalo y así calcular la raíz. Si no hay, devuelve NIL.

Si hay, obtiene los dos primeros elementos de la lista (por ejemplo, lst[i] y lst[i+1]) y realiza el producto ente f(lst[i]) y f(lst[i+1]), siendo f la función para obtener la raíz. Si dicho producto es negativo, hará la bisección en el intervalo [i, i+1].

En caso contrario, se desplazará en la lista de forma recursiva y seguirá buscando intervalos sobre los que obtener raíces. Todas las raíces que encuentre `allroot` se devolverán en forma de lista.

Apartado 2.3

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; make-interval-list (a b i)
;;;
;;; Crea la lista de números entre a y b a distancia i
;;;
;;; INPUT:          a: Mínimo del intervalo
;;;                 b: Máximo del intervalo
;;;                         i: Distancia entre elementos
;;; OUTPUT: lista de números entre a y b a distancia i
;;;
(defun make-interval-list (a b i)
  ;En la comparación usamos b+i/2 como un sobrepaso a errores por decimales
  ;que hacía que a, al llegar a b, fuera ligeramente mayor que b y no
  ;registrase el último número de la lista
  (if (> a (+ b (/ i 2)))
```

```
    nil
  (cons a (make-interval-list (+ a i) b i))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; allind (f a b N tol)
;;;
;;; Calcula todas las raices de una función en los valores establecidos
;;; entre los subintervalos de un intervalo dado.
;;;
;;; INPUT:          f: Función de la que calcular las raices
;;;                 a: Mínimo del intervalo principal
;;;                         b: Máximo del intervalo principal
;;;                         N: Valor que, siendo exponente de 2, se usará para divir el inervalo
;;;                                 principal en sub-intervalos
;;;                         tol: Resultado mínimo para aceptar un resultado
;;; OUTPUT: Listado de las raices de la función en los intervalos establecidos
;;;
(defun allind (f a b N tol)
 ; Aunque se podría itroducir directamente el contenido de interval-lst en allroot,
 ; hemos optado por crear una variable local para aumentar la claridad del código.
 ; A.K.A.: #NoALosChurros
 (let ((interval-lst (make-interval-list a b (/ (+ (abs b) (abs a)) (expt 2 N)))))
  (allroot f interval-lst tol)))
```

Para encontrar todas las raíces de las secciones de un intervalo, la función `allind (f a b N tol)` se sirve de la función auxiliar `make-interval-list (a b i)`.

Esta función auxiliar crea una lista de secciones que abarcan desde a hasta b, pasando por sucesivos incrementos de (a +i) hasta alcanzar b. Es decir:

$$[a\ (a+i)\ (a+i+i)\ (a+i+i+i)\ \ldots\ b]$$

En cuanto a `allind (f a b N tol)`, la función divide el intervalo [a, b] en 2^N secciones mediante la operación $(|a| + |b|) / 2^N$ y calcula todas las raíces de las secciones mediante una llamada a `allroot`.

3. Combinación de listas

Apartado 3.1

Código

```
;;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst (elt lst)
;;;
;;; Genera un listado con la combinación de un átomo con todos los
;;; átomos de un listado.
;;;
;;; INPUT:          elt: Átomo a combinar
;;;                 lst: Listado a combinar
;;; OUTPUT: Listado con todas las combinaciones del átomo y los elementos del listado
;;;
(defun combine-elt-lst (elt lst)
  (if (null lst)
      nil
    (cons (list elt (first lst)) (combine-elt-lst elt (rest lst)))))
```

Para combinar un átomo elt con cada elemento de una lista y crear un listado de cada combinación, la función `combine-elt-lst (elt lst)` sigue estos pasos.

Si lst es NIL (ha llegado al final de la lista para combinar), devolverá NIL. En caso contrario, creará una lista de pares (elt <elemento-lista>) de forma recursiva.

Apartado 3.2

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst (lst1 lst2)
;;;
;;; Genera un listado con el producto cartesiano de elementos de dos
;;; listas dadas.
;;;
;;; INPUT:          lst1: Lista a combinar
;;;                 lst2: Lista a combinar
;;; OUTPUT: Listado con el producto cartesiano de los elementos de ambos listados
;;;
(defun combine-lst-lst (lst1 lst2)
  (unless (or (null lst1) (null lst2))
    nil
    (append (combine-elt-lst (first lst1) lst2) (combine-lst-lst (rest lst1) lst2))))
```

Para crear el producto cartesiano de dos listas, `combine-lst-lst (lst1 lst2)` comprueba primero si lst1 o lst2 es NIL. Si se cumple la condición, devolverá NIL.

Sino, creará una lista de pares (<elemento-lst1> <elemento-lst2>) con cada elemento de lst1 y la lista lst2, de forma recursiva. La función terminará cuando no haya más elementos de lst1 que combinar.

Apartado 3.3

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lists-rec-inner(elt lst)
;;;
;;; Une todas las sublistas de lst con el elemento lst.
;;;
;;; INPUT:          elt: único elemento encapsulado en una lista
;;;                 lst: lista con sublistas
;;; OUTPUT: Listado con el producto cartesiano de los elementos de los sublistados
;;;
(defun prod-cartesiano-inner(elt lst)
  (if (null lst)
      nil
    (cons (append elt (first lst)) (prod-cartesiano-inner elt (rest lst)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lists-rec (lst1 lst2)
;;;
;;; Une cada elemento de lst1 a todos los elementos de lst2, generando,
;;; efectivamente, el producto cartesiano de ambas.
;;;
;;; INPUT:          lst1: lista con sublistas sobre la que iterar elemento a elemento
;;;                 lst2: lista con sublistas sobre la que iterar integramente
;;; OUTPUT: Listado con el producto cartesiano de los elementos de los sublistados
;;;
(defun prod-cartesiano (lst1 lst2)
  (if (or (null lst1) (null lst2))
      nil
    (append (prod-cartesiano-inner (first lst1) lst2)
        (prod-cartesiano (rest lst1) lst2))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lst (lstolsts)
;;;
;;; Genera el producto cartesiano de los elementos de los sublistados
;;; facilitados mediante un listado.
;;;
;;; INPUT:  lstolsts: listado con los sublistados
;;; OUTPUT: Listado con el producto cartesiano de los elementos de los sublistados
;;;
(defun combine-list-of-lsts (lstolsts)
  (if (null (rest lstolsts))
      ;Dejamos como caso base la generación de sublistas de los elementos de una lista dada
      (mapcar #'list (first lstolsts))
    ;Como iteración, tenemos dos funciones recursivas, a las que introduciremos cada par de listas
    ;cuyos elementos ya han sido divididos en sublistas
    (prod-cartesiano (mapcar #'list (first lstolsts))
                     (combine-list-of-lsts (rest lstolsts)))))
```

Para crear el producto cartesiano de un conjunto de sublistas encapsulado en una lista, primero tomamos el caso base de aplicar un map de la función list sobre la última sublista, lo que hace que ya, de base, devuelve el producto cartesiano de esa única sublista.

Seguidamente, mediante dos funciones recursivas, una anidada en la otra, vamos generando el producto cartesiano de ambas listas, básicamente, uniendo, elemento a elemento de la primera lista, a cada uno de los elementos de la segunda.

4. Inferencia lógica proposicional

Apartado 4.1

Apartado 4.1.1

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.1
;; Predicado para determinar si una expresion en LISP
;; es un literal positivo
;;
;; RECIBE   : expresion
;; EVALUA A : T si la expresion es un literal positivo,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun positive-literal-p (x)
 ;Comprueba si x es un atomo y no se ha declarado
 ;como conector, o como NIL o T directamente
 (and (atom x) (not (equal x nil)) (not (equal x t)) (not (connector-p x))))
```

Para comprobar si un literal es positivo, la función `positive-literal-p (x)` comprueba que x sea un átomo, nunca NIL, T ni cualquier tipo de conector.

Apartado 4.1.2

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.2
;; Predicado para determinar si una expresion
;; es un literal negativo
;;
;; RECIBE   : expresion x
;; EVALUA A : T si la expresion es un literal negativo,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun negative-literal-p (x)
 ;Comprueba si x es una lista de 2 elementos,
 ;siendo el primero el conector NOT
 ;y el segundo un literal positivo
 (and (listp x) (unary-connector-p (first x)) (positive-literal-p (second x))))
```

Para comprobar si un literal es negativo, la función `negative-literal-p (x)` comprueba que x sea una lista con un conector ~ y un literal positivo x.

Es decir, (~ x).

Apartado 4.1.3

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.3
;; Predicado para determinar si una expresion es un literal
;;
```

```
;; RECIBE   : expresion x
;; EVALUA A : T si la expresion es un literal,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun literal-p (x)
 ;Comprueba si x es un literal positivo o negativo
 (or (positive-literal-p x) (negative-literal-p x)))
```

Para comprobar si el argumento es un literal, la función `literal-p (x)` comprueba que x sea literal positivo o negativo.

Apartado 4.1.4

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.4
;; Predicado para determinar si una expresion esta en formato prefijo,
;; bien (~ FBF) o (FBF1 <conector> FBF2)
;;
;; RECIBE   : expresion x
;; EVALUA A : T si x esta en formato prefijo,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun wff-infix-p (x)
 (unless (null x)    ;NIL no es FBF en formato infijo (por convencion)
  (or (literal-p x)  ;Un literal es FBF en formato infijo
      (and (listp x)  ;En caso de que no sea un literal debe ser una lista
          (cond
           ;Si el primer elemento es un connector unario ~
           ;debería tener la estructura (<conector> FBF)
           ((unary-connector-p (first x))
            (and (null (rest (rest x)))      ;Después de FBF no hay "nada" (NIL)
                 (wff-infix-p (second x))))
           ;Si el elemento es un conector binario =>, <=>
           ;deberia tener la estructura (FBF1 <conector> FBF2)
           ((binary-connector-p (second x))
            (and (null (fourth x))                    ;Después de FBF2 no hay "nada" (NIL)
                 (wff-infix-p (first x))
                 (wff-infix-p (third x))))
           ;Si el elemento es una conjuncion o disyuncion vacía,
           ;(^) (v), no tienen ninguna FBF detrás (NIL)
           ((n-ary-connector-p (first x))
            (null (rest x)))
           ;Si el elemento es un conector enario ^ ó v, debería
           ;tener la estructura (FBF1 <conector> FBF2 <conector> ... FBFn)
           ((n-ary-connector-p (second x))
            ;Si llegamos a una estructura de tipo (FBFn-1 <conector> FBFn)
            (if (null (fourth x))
                (and (wff-infix-p (first x)) ;Comprueba si FBF1 es válida
                     (wff-infix-p (third x)))  ;Comprueba si FBF2 es válida
              ;En caso de tener una estructura de tipo (FBFi <conector> FBFi+1 <conector> ... FBFn)
              (and (eql (second x) (fourth x))       ;Comprueba si en la estructura no hay dos conectores
distintos como: (A ^ B v C)
                   (wff-infix-p (first x))           ;Comprueba si FBF1 es válida
                   (wff-infix-p (rest (rest x))))))  ;Comprueba si (FBFi+1 <conector> ... FBFn) es válida
           ;No es FBF en formato infijo
           (t NIL))))))
```

Para comprobar si una expresión está en formato prefijo (~ FBF) o (FBF <conector> FBF), la función `wff-infix-p (wff)` sigue una serie de pasos.

Primero comprueba que wff no sea NIL, ya que NIL no es una FBF en formato prefijo. Si wff no es NIL, o bien es un literal o bien es una lista de estos tipos:

- **Lista (~ FBF).** Si tras FBF no hay más elementos, comprueba si FBF está en formato infijo de forma recursiva.

- **Lista (FBF1 <conector-bicondicional> FBF2).** Si después de FBF2 no hay más elementos, comprueba si FBF1 y FBF2 están en formato prefijo de forma recursiva.

- **Lista (<conector-nario>).** Comprueba si se trata de una conjunción o disyunción vacías (‘(v) o ‘(^)).

- **Lista (FBF1 <conector-nario> FBF2 <conector-nario> ... FBFn).** Si la lista es de tipo (FBF1 <conector-nario> FBF2), comprueba si FBF1 y FBF2 están en formato prefijo de forma recursiva.

  Si la lista tiene más FBFs, comprueba si tanto FBF1 como (FBF2 ... FBFn) están en formato prefijo de forma recursiva.

Apartado 4.1.5

Código

```
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.5
;;
;; Convierte FBF en formato infijo a FBF en formato prefijo
;;
;; RECIBE   : FBF en formato infijo
;; EVALUA A : FBF en formato prefijo
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun infix-to-prefix (wff)
 (when (wff-infix-p wff)              ;Mientras wff sea una FBF
  (if (literal-p wff)
    wff                                ;Caso base: wff es un literal
   (cond
    ;Si el primer elemento es un connector unario ~
    ;debería tener la estructura (<conector> FBF)
    ((unary-connector-p (first wff))
     (list (first wff) (infix-to-prefix (second wff))))
    ;Si el elemento es un conector binario =>, <=>
    ;deberia tener la estructura (FBF1 <conector> FBF2)
    ((binary-connector-p (second wff))
     (list (second wff)
          (infix-to-prefix (first wff))
          (infix-to-prefix (third wff))))
    ; Si el elemento es una conjuncion o disyuncion
    ; vacía, se evalúa direcamente.
    ((n-ary-connector-p (first wff))
     (when (null (rest wff))
      wff))
    ;Si el elemento es un conector enario ^ v, debería
    ;tener la estructura (FBF1 <conector> FBF2 <conector> ... FBFn)
    ((n-ary-connector-p (second wff))
```

```
;Crea una lista de tipo (<conector> FBF1 FBF2 ... FBFn)
;sobre una copia de wff en la que todos los conectores
;se han eliminado.
;Por tanto, mapcar evalua cada FBF de la lista y
;devuelve su forma prefijo.
(append (list (second wff))
      (mapcar #'infix-to-prefix (remove (second wff) wff))))
(t NIL)))))
```

Para transformar una FBF en formato infijo a una FBF en formato prefijo, la función `infix-to-prefix-p (wff)` sigue una serie de pasos.

Mientras que wff sea una FBF en formato infijo, comprueba si es un literal. Si lo es, devuelve su valor.

Si wff tiene más elementos (es decir, es una lista de FBFs unidas por conectores), identifica el tipo de conector que las une.

- **Conector unario ~.** Crea una lista con el conector y la FBF convertida a formato infijo de forma recursiva.

- **Conector binario => o <=>.** Crea una lista de tipo (<conector-binario> FBF1 FBF2).

- **Conector n-ario ^ o v.** Si encuentra una lista de tipo '(v) o '(^), devuelve a conjunción o disyunción vacías.

  Si se trata de una lista de tipo (FBF1 <conector-nario> FBF2 <conector-nario> FBFn), crea una nueva lista con el conector n-ario seguido del resto de FBFs. Todas ellas se han convertido a formato prefijo mediante la función:
  `(mapcar #'infix-to-prefix (remove (second wff) wff)`

  La llamada a remove elimina los conectores sobrantes de la lista de FBFs, dando como resultado (FBF2 FBF3 ... FBFn), lista que se evaluará elemento a elemento.

Apartado 4.1.6

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.6
;; Predicado para determinar si una FBF es una clausula
;;
;; RECIBE   : FBF en formato prefijo
;; EVALUA A : T si FBF es una clausula, NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun clause-p (wff)
  (when (listp wff)
   (let ((conector (first wff))
```

```
      (elems (rest wff)))
    (if (eql +or+ conector)
      (or (null elems)                        ;Disyunción vacía (v)
        (and (null (rest elems))              ;Disyunción con un elemento (v lit)
          (literal-p (second wff)))
        (and (literal-p (second wff)) ;Disyunción con más de un elemento
          (clause-p (cons conector (rest elems))))))))
```

La función `clause-p (wff)` determina si una FBF es una cláusula mediante estos criterios:

- El argumento wff es una lista precedida por el conector v.
- La lista contiene una disyunción vacía '(v), una disyunción con un literal '(v lit) o una disyunción con más de un literal. En este último caso se comprueba si cada elemento es un literal de forma recursiva.

Apartado 4.1.7

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.7
;; Predicado para determinar si una FBF esta en FNC
;;
;; RECIBE   : FFB en formato prefijo
;; EVALUA A : T si FBF esta en FNC con conectores,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun cnf-p (wff)
  (when (listp wff)
    (let ((conector (first wff))
        (elems (rest wff)))
      (if (eql +and+ conector)
        (or (null elems)                      ;Conjunción vacía (^)
          (and (null (rest elems))            ;Conjunción de tipo (^ (v)) ó (^ (v clause))
            (clause-p (second wff)))
          (and (clause-p (second wff))        ;Conjunción con más de una cláusula
            (cnf-p (cons conector (rest elems))))))))
```

La función `cnf-p (wff)` determina si una FBF es una FNC mediante estos criterios:

- El argumento wff es una lista precedida por el conector ^.
- La lista contiene una disyunción vacía '(^), una conjunción de tipo '(^ (v)) o '(^ (v lit1 lit2 ... litn) , o una conjunción con más de una cláusula. En este último caso se comprueba si cada elemento es una cláusula de forma recursiva.

Apartado 4.2

Apartado 4.2.1

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.1: Incluya comentarios en el codigo adjunto
;;
;; Dada una FBF, evalua a una FBF equivalente
;; que no contiene el connector <=>
;;
;; RECIBE   : FBF en formato prefijo
;; EVALUA A : FBF equivalente en formato prefijo
;;          sin connector <=>
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-biconditional (wff)
 ;Si la FBF es NIL o un literal, devuelve su valor.
 (if (or (null wff) (literal-p wff))
    wff
           ;Sino, tenemos una estructura de tipo (<conector> FBFs).
  (let ((connector (first wff)))
   (if (eq connector +bicond+)
                        ;Si el conector es <=>, extrae FBF1 y FBF2 y crea
                        ;una lista de tipo (^ (=> FBF1 FBF2) (=> FBF2 FBF1)).
       (let ((wff1 (eliminate-biconditional (second wff)))
          (wff2 (eliminate-biconditional (third  wff))))
        (list +and+
            (list +cond+ wff1 wff2)
            (list +cond+ wff2 wff1)))
                        ;En caso contrario, devuelve una lista de tipo
                        ;(<conector> <FBFs sin <=>>)
       (cons connector
         (mapcar #'eliminate-biconditional (rest wff)))))))
```

Para convertir la FBF pasada como argumento a una FBF equivalente sin el conector <=>, la función `eliminate-biconditional (wff)` sigue una serie de pasos.

En primer lugar, si wff es NIL o un literal devuelve su valor. Si se trata de una lista, tendrá la estructura (<conector> FBF1 FBF2).

Si el conector es bicondicional, trasforma FBF1 y FBF2 a FBFs equivalentes sin <=> de forma recursiva, y crea una lista de tipo (^ (=> FBF1 FBF2) (=> FBF2 FBF1)).

Si el conector es unario o enario, devuelve la misma estructura de FBFs (~ FBF) o (FBF1 <conector-nario> FBF2 <conector-nario> ... FBFn), sin el conector <=>. Cada FBF se evalúa con la función mapcar.

Apartado 4.2.2

Código

```
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.2
;; Dada una FBF, que contiene conectores => evalua a
;; una FBF equivalente que no contiene el connector =>
;;
;; RECIBE   : wff en formato prefijo sin el connector <=>
;; EVALUA A : wff equivalente en formato prefijo
;;          sin el connector =>
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-conditional (wff)
 ;Si la FBF es NIL o un literal, devuelve su valor.
```

```
(if (or (null wff) (literal-p wff))
          wff
         ;Sino, tenemos una estructura de tipo (<conector> FBFs).
         (let ((connector (first wff)))
          (if (eql connector +cond+)
                    ;Si el conector es =>, extrae FBF1 y FBF2 y crea
                    ;una lista de tipo (v (~ FBF1) FBF2).
              (let ((wff1 (eliminate-conditional (second wff)))
                                        (wff2 (eliminate-conditional (third wff))))
                        (list +or+
                                  (list +not+ wff1)
                                  wff2))
                    ;En caso contrario, devuelve una lista de tipo
                    ;(<conector> <FBFs sin =>>).
                    (cons connector
                              (mapcar #'eliminate-conditional (rest wff)))))))
```

Para convertir la FBF pasada como argumento a una FBF equivalente sin el conector =>, la función `eliminate-conditional (wff)` sigue una serie de pasos.

Si wff es NIL o un literal, devuelve su valor. Si se trata de una lista, tendrá la estructura (<conector> FBF1 FBF2).

Si el conector es condicional, trasforma FBF1 y FBF2 a FBFs equivalentes sin => de forma recursiva, y crea una lista de tipo (v (~ FBF1) FBF2).

Si el conector es unario o enario, devuelve la misma estructura de FBFs (~ FBF) o (FBF1 <conector-nario> FBF2 <conector-nario> ... FBFn), sin el conector =>. Cada FBF se evalúa con la función mapcar.

Apartado 4.2.3

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.3
;; exchange-and-or
;;
;; Función auxiliar que sustituye el conector +and+ (^)
;; por not (v) y viceversa.
;;
;; RECIBE   : Conector lógico
;; EVALUA A : Conector or (si recibió and), conector and
;; (si recibió or), o cualquier otro conector pasado como
;; parámetro.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun exchange-and-or (connector)
  (cond
   ((eq connector +and+) +or+)
   ((eq connector +or+) +and+)
   (t connector)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.3
;; Dada una FBF, que no contiene los conectores <=>, =>
;; evalua a una FNC equivalente en la que la negacion
;; aparece unicamente en literales negativos
;;
```

```
;; RECIBE   : FBF en formato prefijo sin conector <=>, =>
;; EVALUA A : FBF equivalente en formato prefijo en la que
;;           la negacion  aparece unicamente en literales
;;           negativos.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun reduce-scope-of-negation (wff)
  ;Si la FBF es NIL o un literal, devuelve su valor.
  (if (or (null wff) (literal-p wff))
     wff
   ;Sino, tenemos una estructura de tipo (<conector> FBFs).
   (let ((connector (first wff))
        (elems (second wff)))
     (if (eql connector +not+)
        ;Si recibe una estructura de tipo (~ (<conector> FBFs)):
        (let ((connector_2 (first (second wff))))
         (cond
          ;CASO 1: FBF de tipo (~ (<conector n-ario> FBF1 FBF2 ... FBFn)).
          ;Primero añade un conector ~ a cada FBF en (FBF1 FBF2 ... FBFn)
          ;y después crea una lista de la forma (<conector n-ario cambiado> <FBFs reducidas en ~>)
          ((n-ary-connector-p connector_2)
           (let ((negated_wffs (mapcar #'(lambda(x) (list +not+ x)) (rest elems)))) ;Niega todas las FBFs.
             (cons (exchange-and-or connector_2)
                (mapcar #'reduce-scope-of-negation negated_wffs))))       ;Por cada FBF negada, reduce el
ámbito del conector ~.
          ;CASO 2: FBF de tipo (~ (~ FBF))
          ;Evalúa y reduce el ámbito de ~ en FBF.
          ((eql connector_2 +not+)
           (reduce-scope-of-negation (second elems)))
          (t NIL)))
      ;Si la estructura es de tipo (<conector n-ario> FBF1 FBF2 ... FBFn),
      ;evalúa cada una de las FBFs en caso de que se necesite reducir
      ;el ámbito de ~.
      (cons connector
         (mapcar #'reduce-scope-of-negation (rest wff))))))
```

La función `reduce-scope-of-negation (wff)` convierte una FBF sin conectores bicondicionales a una FBF equivalente donde la negación se realiza a nivel de literal.

Primero, la función comprueba si wff es NIL o un literal. Si lo es, devuelve su valor.

En caso de que wff sea una lista, la función considerará los siguientes casos:

- **Lista de tipo (~ (<conector-nario> FBFs).** Niega cada FBF de la lista mediante la función:

  `(mapcar #'(lambda(x) (list +not+ x)) (rest elems))`

  Y construye una lista con estos elementos:

  1. Conector n-ario ^ cambiado por v (o al revés) llamando a la función auxiliar `exchange-and-or (connector)`.
  2. FBFs negadas previamente con el ámbito de negación reducido. El ámbito de la negación de cada una de ellas se reduce mediante la función mapcar.

- **Lista de tipo (~ (~ FBF)).** Devuelve la FBF con el ámbito de negación reducido. Para ello utiliza una llamada recursiva a `reduce-scope-of-negation`.

- **Lista de tipo (<conector-nario> FBFs).** Crea una lista con el conector n-ario y cada una de las FBFs reducidas en ámbito de negación mediante la función mapcar.

Apartado 4.2.4

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.4: Comente el codigo adjunto
;;
;; Dada una FBF, que no contiene los conectores <=>, => en la
;; que la negacion aparece unicamente en literales negativos
;; evalua a una FNC equivalente en FNC con conectores ^, v
;;
;; RECIBE   : FBF en formato prefijo sin conector <=>, =>,
;;          en la que la negacion aparece unicamente
;;          en literales negativos
;; EVALUA A : FBF equivalente en formato prefijo FNC
;;          con conectores ^, v
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Función que crea listas de listas, siendo
;; cada una de tipo (elt <elemento-lst>
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun combine-elt-lst (elt lst)
 ;Si la lista es NIL, crea una lista
 ;de tipo ((elem))
 (if (null lst)
    (list (list elt))
          ;Si no es NIL, combina elt con cada elemento
          ;lst de esta forma:
          ;(elt elem-lst-1) (elt-elem-lst-2)... (elt elem-lst-n)
  (mapcar #'(lambda (x) (cons elt x)) lst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Función que añade conectores a una FBF
;; con elementos combinados entre sí
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun exchange-NF (nf)
 ;Si la FBF es NIL o un literal, devuelve su valor.
 (if (or (null nf) (literal-p nf))
    nf
          ;Sino, construye una lista con el conector n-ario
          ;cambiado y las sublistas (<conector n-ario normal> FBF);
          ;todo ello utilizando la FBF combinada.
  (let ((connector (first nf)))
   (cons (exchange-and-or connector)
      (mapcar #'(lambda (x)
                (cons connector x))
         (exchange-NF-aux (rest nf)))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Función que combina los elementos de una FBF
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun exchange-NF-aux (nf)
```

```
 ;Si recibe NIL como argumento, devuelve NIL.
 (if (null nf)
   NIL
         ;Sino, obtiene el primer elemento de la FBF.
         ;
         ;Si el primer elemento es un literal, construirá
         ;una lista combinándolo con el resto de la FBF.
         ;Si no es un literal, combinará el resto de la FBF
         ;con sus propios elementos.
  (let ((lst (first nf)))
   (mapcan #'(lambda (x)
         (combine-elt-lst
          x
          (exchange-NF-aux (rest nf))))
    (if (literal-p lst) (list lst) (rest lst))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Función que simplifica una FBF
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun simplify (connector lst-wffs )
 ;Si la FBF pasada como argumento es un literal,
 ;devuelve su valor
 (if (literal-p lst-wffs)
   lst-wffs
         ;Sino, crea una lista simplificada sobre lst-wffs
         ;dependiendo de ciertos criterios.
  (mapcan #'(lambda (x)
        (cond
                              ;Si el elemento es un literal, lo añade
                              ;a una lista.
         ((literal-p x) (list x))
                              ;Si el primer elemento de la FBF coincide
                              ;con el conector pasado como argumento,
                              ;simplifica el resto de la lista.
         ((equal connector (first x))
          (mapcan
            #'(lambda (y) (simplify connector (list y)))
           (rest x)))
                              ;Si el elemento no es un literal (con lo que
                              ;sería un conector) y no coincide con el conector
                              ;pasado como argumento, mete el elemento en una lista.
         (t (list x))))
   lst-wffs)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Función que traduce una FBF a FNC
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun cnf (wff)
 (cond
  ;Si wff es una FNC, devuelve su contenido.
  ((cnf-p wff) wff)
  ;Si wff es un literal, crea una lista
  ;de tipo (^ (v lit)) (FNC con 1 elemento).
  ((literal-p wff)
   (list +and+ (list +or+ wff)))
  ((let ((connector (first wff)))      ;Evalúa el conector de wff.
   (cond
          ;CASO 1: Conector ^
          ;Simplifica y evalúa el resto de la FBF.
    ((equal +and+ connector)
     (cons +and+ (simplify +and+ (mapcar #'cnf (rest wff)))))
          ;CASO 2: Conector v
          ;<Intercambia> la FBF simplificada previamente.
    ((equal +or+ connector)
     (cnf (exchange-NF (cons +or+ (simplify +or+ (rest wff))))))))))
```

Apartado 4.2.5

Código

```
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.5:
;;
;; Dada una FBF en  FNC
;; evalua a lista de listas sin conectores
;; que representa una conjuncion de disyunciones de literales
;;
;; RECIBE   : FBF en FNC con conectores ^, v
;; EVALUA A : FBF en FNC (con conectores ^, v eliminaos)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-connectors (cnf)
 ;Si cnf es un literal, devuelve su valor.
 (if (literal-p cnf)
    cnf
  ;Sino, comprueba si el primer elemento es el
  ;conector ^ ó v.
  (let ((connector (first cnf)))
    (when (n-ary-connector-p connector)
      ;Siempre que se cumpla la condición, evaluará
      ;cada elemento de cnf y devolverá una lista de
      ;listas sin conectores.
      (mapcar #'eliminate-connectors (rest cnf))))))
```

Para eliminar los conectores de una FNC, la función `eliminate-connectors (cnf)` comprueba primero si la FNC pasada como argumento es un literal. En ese caso devolverá su valor.

Sino, cada elemento de la FNC que contenga un conector n-ario se evaluará mediante la función:

```
(mapcar #'eliminate-connectors (rest cnf))
```

Y se convertirá a una lista de literales sin conectores.

Apartado 4.2.6

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.6
;; FUNCIONES AUXILIARES
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Función que devuelve la FBF
;; convertida a formato prefijo
(defun transformar-prefijo (wff)
 (infix-to-prefix wff))

;; Función que devuelve la FBF en
;; formato prefijo sin conectores <=>
;; (Paso 1)
```

```
(defun eliminar-bicond (wff)
  (let ((wff-paso-1 (transformar-prefijo wff)))
    (eliminate-biconditional wff-paso-1)))

;; Función que devuelve la FBF en
;; formato prefijo sin conectores =>
;; (Paso 2)
(defun eliminar-cond (wff)
  (let ((wff-paso-2 (eliminar-bicond wff)))
    (eliminate-conditional wff-paso-2)))

;; Función que devuelve la FBF en
;; formato prefijo con el ámbito
;; de negación reducido
;; (Paso 3)
(defun reducir-negacion (wff)
  (let ((wff-paso-3 (eliminar-cond wff)))
    (reduce-scope-of-negation wff-paso-3)))

;; Función que devuelve la FBF en
;; formato prefijo convertida a FNC
;; (Paso 4)
(defun get-cnf (wff)
  (let ((wff-paso-4 (reducir-negacion wff)))
    (cnf wff-paso-4)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.6
;; Dada una FBF en formato infijo
;; evalua a lista de listas sin conectores
;; que representa la FNC equivalente
;;
;; RECIBE   : FBF
;; EVALUA A : FBF en FNC (con conectores ^, v eliminados)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun wff-infix-to-cnf (wff)
  ;Siempre que wff sea FNC, obtendrá
  ;su forma FBF sin conectores.
  (when (wff-infix-p wff)
    (let ((cnf (get-cnf wff)))
      (eliminate-connectors cnf))))
```

Para convertir una FBF en formato infijo a una FNC en formato prefijo que no tenga conectores, la función `wff-infix-to-cnf (wff)` realiza llamadas a funciones auxiliares que van transformando la FNC paso a paso como se indica en los apartados anteriores.

Apartado 4.3

Apartado 4.3.1

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.1
;; eliminacion de literales repetidos una clausula
;;
;; RECIBE   : K - clausula (lista de literales, disyuncion implicita)
;; EVALUA A : clausula equivalente sin literales repetidos
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-repeated-literals (k)
  (let ((lit (first k))
        (elems (rest k)))
    (cond
     ;CASO1: Hemos llegado al final.
     ((null elems) k)
     ;CASO 2: Un elemento tiene sólo una ocurrencia.
     ;Para comparar, utilizamos su representación textual (equal).
     ((= (count lit k :test #'equal) 1)
      (cons lit (eliminate-repeated-literals (rest k))))
     ;CASO 3: Un elemento tiene más de una ocurrencia.
     (t (eliminate-repeated-literals (rest K))))))
```

La función `eliminate-repeated-literals (k)` transforma la cláusula K a una cláusula equivalente sin literales repetidos siguiendo una serie de pasos.

En primer lugar, comprueba si K sólo tiene un elemento. Si se cumple esta condición, devuelve el literal de K (ya sea positivo o negativo). En cambio, si K tiene más de un elemento, la función comprueba cuántas ocurrencias de este elemento encuentra en K.

- **Sólo hay una ocurrencia.** Crea una lista con el único literal y el resto de los elementos de K, filtrados de forma recursiva.

- **Hay más de una ocurrencia.** El elemento se ignora y se evalúa el resto de los literales de K de forma recursiva.

Apartado 4.3.2

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.2
;; FUNCIONES AUXILIARES
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; check-if-equal (cls target-cls)
;;;
;;; Comprueba si una cláusula es igual que otra que está en
;;; la FNC, se representen de manera idéntica o diferente.
;;; Por ejemplo, '(a b c) = '(b a c)
;;;
;;; INPUT: cls: cláusula 1 de la comprobación.
;;;              target-cls: cláusula 2 de la comprobación.
;;; OUTPUT: T si son iguales, NIL si no.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun check-if-equal (cls target-cls)
  ;O las cláusulas se representan textualmente
  ;de igual forma (por ejemplo, '(a b c) '(a b c))
  (or (equal cls target-cls)
      ;O, pese a representarse de forma distinta,
      ;ambas contienen los mismos elementos
      ;(por ejemplo, '(a b c) '(b a c)).
      (and (null (set-difference cls target-cls :test #'equal))
           (null (set-difference target-cls cls :test #'equal)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; check-if-repeated-in-cnf (clause cnf)
;;;
;;; Comprueba si la cláusula clause está repetida en
;;; la FNC pasada como argumento.
;;;
;;; INPUT: clause: cláusula que se va a comprobar.
;;;              cnf: FNC que puede o no contener cláusulas repetidas.
;;; OUTPUT: T si hay coincidencias, NIL si no.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun check-if-repeated-in-cnf (clause cnf)
  ;Si hemos llegado al final de la FNC,
  ;clause no está repetida.
  (if (null cnf)
      nil
    ;Si aún no hemos llegado al final:
    (or (check-if-equal clause (first cnf))                    ;O coinciden clause y el primer elemento de cnf.
        (check-if-repeated-in-cnf clause (rest cnf))))) ;O hay coincidencias con alguno del resto

                              ;de elementos de cnf.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-cnf-without-repeated-clauses (cnf)
;;;
;;; Devuelve una FNC sin cláusulas repetidas.
;;;
;;; INPUT: cnf: FNC que puede o no contener cláusulas repetidas.
;;; OUTPUT: FNC sin cláusulas repetidas.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun get-cnf-without-repeated-clauses (cnf)
  (cond
    ;Si hemos llegado al final de FNC,
    ;no hay más cláusulas repetidas que filtrar.
    ((null cnf)
     nil)
    ;Comprueba si la cláusula coincide con alguno de los
    ;elementos de la FNC. Si coincide, avanza en la FNC.
    ((check-if-repeated-in-cnf (first cnf) (rest cnf))
     (get-cnf-without-repeated-clauses (rest cnf)))
    ;Si no hay coincidencias, crea una lista con el
    ;elemento único y el resto de la FNC filtrada.
    (t (cons (first cnf)
             (get-cnf-without-repeated-clauses (rest cnf))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.2
;; eliminacion de clausulas repetidas en una FNC
;;
;; RECIBE   : cnf - FBF en FNC (lista de clausulas, conjuncion implicita)
;; EVALUA A : FNC equivalente sin clausulas repetidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-repeated-clauses (cnf)
  ;Obtiene la FNC sin literales repetidos en sus cláusulas.
  (let ((no-rep-cnf (mapcar #'eliminate-repeated-literals cnf)))
    ;Obtiene la FNC sin cláusulas repetidas.
    (get-cnf-without-repeated-clauses no-rep-cnf)))
```

Para eliminar las cláusulas repetidas de una FNC, la función `eliminate-repeated-clauses (cnf)` elimina los literales repetidos de cada cláusula de la FNC y utiliza una serie de funciones auxiliares.

1. **`check-if-equal (cls target-cls).`** Comprueba si dos cláusulas son iguales, o bien mediante equal (misma representación textual) o bien porque la diferencia de conjuntos entre cls y target-cls (y viceversa) es NIL.

   En este último caso, el valor NIL en las operaciones set-difference indica que ambas cláusulas son iguales, aunque se representen de forma distinta (por ejemplo, '(a b c) y '(b c a)).

2. **`check-if-repeated-in-cnf (clause cnf).`** Comprueba si la cláusula clause está repetida en la FNC pasada como argumento.

   Para ello, comprueba si clause es igual que alguno de los elementos de la FNC de forma recursiva. Si la función llega al final de la FNC (null cnf), no hay coincidencias.

3. **`get-cnf-without-repeated-clauses (cnf).`** Obtiene una FNC sin cláusulas repetidas de forma recursiva.

   Si la función llega al final de la FNC, no hay más cláusulas que evaluar y devuelve NIL. Sino, comprueba si la cláusula tiene repeticiones en la FNC.

   - Tiene repeticiones. La función ignora la cláusula repetida y evalúa el resto de los elementos de la FNC.

   - No tiene repeticiones. La función construye una lista con la cláusula única y el resto de los elementos de la FNC, con sus duplicados eliminados de forma recursiva.

Apartado 4.3.3

Código

```
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.3
;; Predicado que determina si una clausula subsume otra
;;
;; RECIBE   : K1, K2 clausulas
;; EVALUA a : (list K1) si K1 subsume a K2
;;          NIL en caso contrario
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun subsume (K1 K2)
  ;Mientras K1 sea subconjunto de K2, la función
  ;devolverá una lista con K1.
  (when (subsetp K1 K2 :test #'equal)
    (list K1)))
```

Para comprobar si la cláusula K1 subsume a K2, la función `subsume (K1 K2)` comprueba si K1 es un subconjunto de K2 (es decir, si todos los literales de K1 se encuentran en K2).

Si se cumple la condición, devuelve una lista con K1 como elemento.

Apartado 4.3.4

Código

```
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.4
;; eliminacion de clausulas subsumidas en una FNC
;;
;; RECIBE   : cnf (FBF en FNC)
;; EVALUA A : FBF en FNC equivalente a cnf sin clausulas subsumidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-subsumed-clauses (cnf)
 (eliminate-subsumed-clauses-rec cnf cnf))

;; Función que permite a eliminate-subsumed-clauses aplicar
;; recursividad de forma transparente
(defun eliminate-subsumed-clauses-rec (cnf cnf-original)
 (if (null cnf)
    ;; Usamos como caso original la cnf original, dado que nil
    ;; no nos permitiría usar intersection para reconstruir
    ;; la lista resultante
    cnf-original
   (if (null (first-needs-removal? (first cnf) (rest cnf)))
      ;; Si el primer elemento no necesita ser quitado, se une a
      ;; la lista resultante de la limpieza de clausulas subsumidas
      (append
       (list (first cnf))
       (intersection
        (remove-subsumed-clauses (first cnf) (rest cnf))
        (eliminate-subsumed-clauses-rec (rest cnf) cnf-original)
        :test #'equal))
    ;; Si hay que eliminarla, símplemente no se añade al resultado
    (intersection
     (remove-subsumed-clauses (first cnf) (rest cnf))
     (eliminate-subsumed-clauses-rec (rest cnf) cnf-original)
     :test #'equal))))

;; Función que discierne si first-fbf es subsumidas de alguna de
;; las clausulas contenidas en rest-cnf
(defun first-needs-removal? (first-fbf rest-cnf)
 (if (null (first rest-cnf))
    nil
   (if (null (subsume (first rest-cnf) first-fbf))
      (or nil (first-needs-removal? first-fbf (rest rest-cnf)))
    T)))

;; Función que elimina todas las clausulas de rest-cnf que son
;; una subsumiciones de first-fbf
(defun remove-subsumed-clauses (first-fbf rest-cnf)
 (if (null (first rest-cnf))
    '()
   (if (null (subsume first-fbf (first rest-cnf)))
      (append (list (first rest-cnf)) (remove-subsumed-clauses first-fbf (rest rest-cnf)))
```

Para eliminar las clausulas subumidas de una fcn utilizamos una serie de funciones auxiliares para, fundamentalmente, recrear una lista que sólo contiene los elementos no subsumidos en ningún otro elemento de la clausa.

En esta tarea, nos servimos de las siguientes funciones auxiliares:

1. **eliminate-subsumed-clauses (cnf cnf-original)** Para aplicar, de forma transparente, la recursividad sobre nuestra función. Requiere de mantener a lo largo de las llamadas recursivas la lista de clausulas original de cara a que se pueda aplicar la intersección para unir los resultados de funciones auxiliares más internas.

   Así mismo, dada la naturaleza de los bucles internos, como se explicará en unos instantes, que devuelven todas las cápsulas que no tienen la primera clausula de la iteración subsumida, excluyendo así, por supuesto, a la primera clausula en dicha devolución. La adicción de esta al resultado final dependerá de otra función, muy similar a la previamente mencionada para reformar formar la lista sin clausulas subsumidas respecto a la primera clausula de la iteración, sólo que esta simplemente devolverá T si esta primera clausula está subsumida en alguna otra o nil en caso contrario. Así, si recibe un nil, realizará un append de esta primera clausula sobre la lista devuelta por la función interna para mantener el elemento o, en caso contrario, no lo hará para eliminarlo.

2. **first-needs-removal? (first-fbf rest-cnf)** Función auxiliar que, mediante la recursividad de la función *or*, discierne si la clausula first-fbf está subsumida en alguna otra clausula de rest -cnf y, por tanto, debe ser 'eliminada', devolviendo T en ese caso o nil en el contrario.
3. **remove-subsumed-clauses (first-fbf rest-cnf)** Función auxiliar que reforma las listas 'eliminando' las clausulas que tienen la clausula first-fbf subsumida, iterando una de las clausulas de rest-cnf en cada iteración y sólo añadiendo a la lista reformada las clausulas que pasan el criterio.

Apartado 4.3.5

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.5
;; FUNCIONES AUXILIARES
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; literals-with-same-component lit1 lit2)
;;;
;;; Comprueba si 2 literales en forma positiva y negativa
;;; tienen el mismo componente (por ejemplo, p y (~ p))
;;;
;;; INPUT: lit1: literal 1 de la comprobación.
;;;                    lit2: literal 2 de la comprobación.
;;; OUTPUT: t si tienen el mismo componente, NIL si no.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun literals-with-same-component (lit1 lit2)
  (cond
   ;CASO 1: El literal es positivo.
   ;Comprueba si (~ lit1) tiene el mismo componente
```

```
  ;que el literal negativo (~ lit2).
  ((positive-literal-p lit1)
   (equal (list +not+ lit1) lit2))
  ;CASO 2: El literal es negativo.
  ;Comprueba si lit1 tiene el mismo componente
  ;que el literal positivo lit2.
  ((negative-literal-p lit1)
   (equal (second lit1) lit2))
  ;Los literales no tienen el mismo componente.
  (t NIL)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; check-if-pos-and-neg (lit1 K)
;;;
;;; Comprueba si en una cláusula K hay un literal positivo lit1
;;; y otro negativo (~ lit1) o vice versa.
;;;
;;; INPUT: lit1: literal 1 de la comprobación.
;;;                    lit2: literal 2 de la comprobación.
;;; OUTPUT: t si tienen el mismo componente, NIL si no.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun check-if-pos-and-neg (lit K)
 ;Si hemos llegado al final de K,
 ;no hay un literal positivo y otro negativo.
 (if (null K)
    nil
  ;Sino, comprueba que el literal tiene el mismo componente
  ;que el primer elemento de K o que alguno de los elementos
  ;del resto de K.
  (or (literals-with-same-component lit (first K))
     (check-if-pos-and-neg lit (rest K)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.5
;; Predicado que determina si una clausula es tautologia
;;
;; RECIBE   : K (clausula)
;; EVALUA a : T si K es tautologia
;;          NIL en caso contrario
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun tautology-p (K)
 ;Si la cláusula es vacía o tiene sólo un elemento,
 ;su valor de verdad es F.
 (if (or (null K) (null (rest K)))
    nil
  ;Sino, comprueba si hay algún literal positivo
  ;y negativo con el mismo componente en K.
  (or (check-if-pos-and-neg (first K) (rest K))
     (tautology-p (rest K)))))
```

Para comprobar si la cláusula K es una tautología (es decir, si contiene los literales p y (~ p)), la función `tautology-p (K)` se sirve de varias funciones auxiliares.

1. **`literals-with-same-component (lit1 lit2).`** Comprueba si lit1 y lit2 (literales o bien en forma positiva y negativa, o bien en forma negativa y positiva) tienen el mismo literal. Por ejemplo, 'p y '(~ p) tendrían el mismo componente p.

   Para ello, comprueba si lit1 es un literal positivo. Si lo es, comprueba si (~ lit1) es igual que lit2, literal negativo pasado como argumento.

En caso de que lit1 sea negativo (~ lit1), comprueba si lit1 es igual que lit2, literal positivo pasado como argumento.

Si lit1 y lit2 no tienen el mismo componente, la función devolverá NIL.

2. **`check-if-pos-and-neg (lit K).`** Comprueba si en la cláusula K hay un literal positivo lit y otro negativo (~ lit). Para ello, recorre cada literal de K.

    - Si el literal es positivo, comprueba si existe otro negativo con el mismo componente.

    - Si el literal es negativo, comprueba si existe otro positivo con el mismo componente.

    Si la función llega al final de K, ello implica que no ha encontrado dos literales con el mismo componente.

A partir de estas funciones, `tautology-p (K)` comprueba si entre alguno de sus literales hay uno positivo y otro negativo con el mismo componente. Si detecta una cláusula vacía o con un único elemento, devolverá NIL.

Apartado 4.3.6

Código

```
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.6
;; eliminacion de clausulas en una FBF en FNC que son tautologia
;;
;; RECIBE   : cnf - FBF en FNC
;; EVALUA A : FBF en FNC equivalente a cnf sin tautologias
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-tautologies (cnf)
 ;Si hemos llegado al final de la FNC,
 ;no hay más cláusulas que evaluar.
 (if (null cnf)
   nil
  ;Sino, comprueba si cada cláusula de la FNC es una tautología.
  (if (tautology-p (first cnf))
    ;Si lo es, se elimina.
    (eliminate-tautologies (rest cnf))
   ;En caso contrario, construye una lista con la cláusula
   ;que no es tautología y el resto de la FNC.
   (cons (first cnf)
      (eliminate-tautologies (rest cnf)))))
```

Para eliminar las cláusulas de una FNC que son tautología, `eliminate-tautologies (cnf)` comprueba, por cada cláusula, si alguna es tautología.

Si lo es, ignora la cláusula y evalúa el resto de elementos de la FNC de forma recursiva. En caso contrario, construye una lista con la cláusula que no es tautología y el resto de cláusulas de la FNC, también evaluadas de forma recursiva.

Apartado 4.3.7

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.7
;; FUNCIONES AUXILIARES
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Función que devuelve la FNC
;; sin literales repetidos.
(defun fnc-sin-literales-repetidos (cnf)
  (eliminate-repeated-literals cnf))

;; Función que devuelve la FNC
;; sin claúsulas repetidas.
(defun fnc-sin-clausulas-repetidas (cnf)
  (let ((cnf-1 (fnc-sin-literales-repetidos cnf)))
    (eliminate-repeated-clauses cnf-1)))

;; Función que devuelve la FNC
;; sin tautologías.
(defun fnc-sin-tautologias (cnf)
  (let ((cnf-2 (fnc-sin-clausulas-repetidas cnf)))
    (eliminate-tautologies cnf-2)))

;; Función que devuelve la FNC
;; simplificada y sin cláusulas
;; subsumidas.
(defun get-fnc-simplificada (cnf)
  (let ((cnf-3 (fnc-sin-tautologias cnf)))
    (eliminate-subsumed-clauses cnf-3)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.7
;; simplifica FBF en FNC
;;        * elimina literales repetidos en cada una de las clausulas
;;        * elimina clausulas repetidas
;;        * elimina tautologias
;;        * elimina clausulass subsumidas
;;
;; RECIBE   : cnf  FBF en FNC
;; EVALUA A : FNC equivalente sin clausulas repetidas,
;;            sin literales repetidos en las clausulas
;;            y sin clausulas subsumidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun simplify-cnf (cnf)
  ;Obtiene la FNC simplificada aplicando
  ;las operaciones especificadas en
  ;las funciones auxiliares.
  (get-fnc-simplificada cnf))
```

Para simplificar una FNC, la función `simplify-cnf (cnf)` realiza llamadas a funciones auxiliares que van aplicando la eliminación de literales y cláusulas repetidos, la eliminación de tautologías y la eliminación de cláusulas subsumidas como se indica en los apartados anteriores.

Apartado 4.4

Apartado 4.4.1

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.1
;; FUNCIÓN AUXILIAR
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; contains-pos-or-neg-literal (lit clause)
;;;
;;; Comprueba si el literal lit se encuentra en la
;;; cláusula clause, ya sea en forma positiva o negativa.
;;;
;;; INPUT: lit: literal de la comprobación.
;;;             clause: cláusula donde se va a buscar.
;;; OUTPUT: t si el literal se encuentra en clause, NIL si no.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun contains-pos-or-neg-literal (lit clause)
 ;Si clause es NIL, no hay literal que buscar.
 (if (null clause)
   nil
  ;O encuentra el literal lit en forma positiva dentro de clause.
  (or (not (null (member lit clause :test #'equal)))
    ;O encuentra el literal lit en forma negativa dentro de clause.
    (not (null (member (list +not+ lit) clause :test #'equal))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.1
;; Construye el conjunto de clausulas lambda-neutras para una FNC
;;
;; RECIBE   : cnf    - FBF en FBF simplificada
;;            lambda - literal positivo
;; EVALUA A : cnf_lambda^(0) subconjunto de clausulas de cnf
;;            que no contienen el literal lambda ni ~lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun extract-neutral-clauses (lambda cnf)
 ;Si hemos llegado al final de la FNC, no
 ;hay más cláusulas que evaluar.
 (if (null cnf)
   nil
  ;Sino, comprueba si el literal (positivo o negativo)
  ;está en la primera cláusula de la FNC.
  (if (contains-pos-or-neg-literal lambda (first cnf))
    ;Si está, elimina la cláusula.
    (extract-neutral-clauses lambda (rest cnf))
   ;Si no está, construye una lista con la cláusula
   ;y el resto de elementos de la FNC.
   (cons (first cnf)
      (extract-neutral-clauses lambda (rest cnf))))))
```

Para extraer el conjunto de cláusulas lambda-neutras ($\alpha_\lambda^{(0)}$) de una FNC, `extract-neutral-clauses (lambda cnf)` se sirve de la función auxiliar `contains-pos-or-neg-literal (lit clause)`.

Esta función auxiliar comprueba si el literal lit se encuentra en forma positiva o negativa (~ lit) dentro de la cláusula pasada como argumento. Si la cláusula es NIL, la función devolverá NIL (el literal no está).

En cuanto a `extract-neutral-clauses (lambda cnf)`, la función comprueba, cláusula a cláusula, si alguna de ellas tiene algún literal λ positivo o negativo. Las que contengan estos literales se ignorarán, mientras que las que no cumplan la condición se devolverán como una lista de cláusulas neutras.

Apartado 4.4.2

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.2
;; FUNCIÓN AUXILIAR
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; contains-positive-literal (lit clause)
;;;
;;; Comprueba si el literal positivo lit
;;; se encuentra en la cláusula clause.
;;;
;;; INPUT: lit: literal de la comprobación.
;;;                 clause: cláusula donde se va a buscar.
;;; OUTPUT: t si el literal positivo se encuentra en clause,
;;; NIL si no.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun contains-positive-literal (lit clause)
  ;Si clause es NIL, no hay literal que buscar.
  (if (null clause)
      nil
   ;Sino, comprueba si lit se encuentra entre alguno
   ;de los elementos de clause.
   (not (null (member lit clause :test #'equal)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.2
;; Construye el conjunto de clausulas lambda-positivas para una FNC
;;
;; RECIBE   : cnf    - FBF en FNC simplificada
;;            lambda - literal positivo
;; EVALUA A : cnf_lambda^(+) subconjunto de clausulas de cnf
;;            que contienen el literal lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun extract-positive-clauses (lambda cnf)
  ;Si hemos llegado al final de la FNC,
  ;no hay más cláusulas que analizar.
  (if (null cnf)
      nil
   ;Si aún quedan cláusulas, comprueba si no contienen
   ;literales positivos lambda.
   ;En ese caso ignorará la cláusula y avanzará en la FNC.
   (if (null (contains-positive-literal lambda (first cnf)))
       (extract-positive-clauses lambda (rest cnf))
    ;Si hay literales positivos lambda en la FNC,
    ;construye una lista con la cláusula y el resto de
    ;elementos de la FNC.
    (cons (first cnf)
          (extract-positive-clauses lambda (rest cnf))))))
```

Para extraer el conjunto de cláusulas lambda-positivas ($\alpha_{\lambda}^{(+)}$) de una FNC, `extract-positive-clauses (lambda cnf)` se sirve de la función auxiliar `contains-positive-literal (lit clause)`.

Esta función auxiliar comprueba si el literal positivo lit se encuentra dentro de la cláusula pasada como argumento. Si la cláusula es NIL, la función devolverá NIL (el literal no está).

En cuanto a `extract-positive-clauses (lambda cnf)`, la función comprueba, cláusula a cláusula, si alguna de ellas tiene algún literal λ positivo. Todas las cláusulas que contengan estos literales se devolverán como una lista de cláusulas positivas; el resto se ignorará.

Apartado 4.4.3

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.3
;; FUNCIÓN AUXILIAR
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; contains-negative-literal (lit clause)
;;;
;;; Comprueba si el literal lit se encuentra
;;; en forma negativa (~ lit) en la cláusula clause.
;;;
;;; INPUT: lit: literal de la comprobación.
;;;                         clause: cláusula donde se va a buscar.
;;; OUTPUT: t si el literal negativo se encuentra en clause,
;;; NIL si no.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun contains-negative-literal (lit clause)
  ;Si clause es NIL, no hay literal que buscar.
  (if (null clause)
      nil
    ;Sino, comprueba si (~ lit) se encuentra entre alguno
    ;de los elementos de clause.
    (not (null (member (list +not+ lit) clause :test #'equal)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.3
;; Construye el conjunto de clausulas lambda-negativas para una FNC
;;
;; RECIBE   : cnf    - FBF en FNC simplificada
;;          lambda - literal positivo
;; EVALUA A : cnf_lambda^(-) subconjunto de clausulas de cnf
;;          que contienen el literal ~lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun extract-negative-clauses (lambda cnf)
  ;Si hemos llegado al final de la FNC,
  ;no hay más cláusulas que analizar.
  (if (null cnf)
      nil
    ;Si aún quedan cláusulas, comprueba si no contienen
    ;literales negativos (~ lambda).
    ;En ese caso ignorará la cláusula y avanzará en la FNC.
    (if (null (contains-negative-literal lambda (first cnf)))
```

```
      (extract-negative-clauses lambda (rest cnf))
    ;Si hay literales negativos (~ lambda) en la FNC,
    ;construye una lista con la cláusula y el resto de
    ;elementos de la FNC.
    (cons (first cnf)
        (extract-negative-clauses lambda (rest cnf)))))
```

Para extraer el conjunto de cláusulas lambda-negativas ($\alpha_{\lambda}^{(-)}$) de una FNC, `extract-negative-clauses (lambda cnf)` se sirve de la función auxiliar `contains-negative-literal (lit clause)`.

Esta función auxiliar comprueba si el literal negativo (~ lit) se encuentra dentro de la cláusula pasada como argumento. Si la cláusula es NIL, la función devolverá NIL (el literal no está).

En cuanto a `extract-negative-clauses (lambda cnf)`, la función comprueba, cláusula a cláusula, si alguna de ellas tiene algún literal negativo ($\sim \lambda$). Todas las cláusulas que contengan estos literales se devolverán como una lista de cláusulas negativas; el resto se ignorará.

Apartado 4.4.4

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.4
;; FUNCIONES AUXILIARES
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; resolvable-clauses-p (lambda K1 K2)
;;;
;;; Comprueba si dos cláusulas K1 y K2 se pueden resolver.
;;;
;;; INPUT: lambda: literal de la comprobación.
;;;                  K1: cláusula simplificada 1 de la comprobación.
;;;                  K2: cláusula simplificada 2 de la comprobación.
;;; OUTPUT: t si K1 y K2 se pueden resolver, NIL si no.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun resolvable-clauses-p (lambda K1 K2)
 ;O K1 tiene el literal lambda positivo y K2
 ;tiene el literal lambda negativo.
 (or (and (contains-positive-literal lambda K1)
        (contains-negative-literal lambda K2))
    ;O viceversa.
    (and (contains-positive-literal lambda K2)
        (contains-negative-literal lambda K1))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-resolved-clause (lambda K)
;;;
;;; Devuelve una cláusula K resuelta sobre lambda.
;;;
;;; INPUT: lambda: literal sobre el que se va a
;;;       resolver la cláusula.
;;;                  K: cláusula que se va a resolver.
;;; OUTPUT: Cláusula resuelta o NIL si K no contiene a lambda.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun get-resolved-clause (lambda K)
 (cond
  ;CASO 1: K contiene al literal positivo lambda.
  ;Todas las ocurrencias de lambda se eliminan de K.
  ((contains-positive-literal lambda K)
   (remove lambda K :test #'equal))
  ;CASO 2: K contiene al literal negativo (~ lambda).
  ;Todas las ocurrencias de (~ lambda) se eliminan de K.
  ((contains-negative-literal lambda K)
   (remove (list +not+ lambda) K :test #'equal))
  (t NIL))) ;No debería llegar a este caso nunca.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.4
;; resolvente de dos clausulas
;;
;; RECIBE   : lambda     - literal positivo
;;          K1, K2     - clausulas simplificadas
;; EVALUA A : res_lambda(K1,K2)
;;                    - lista que contiene la
;;                      clausula que resulta de aplicar resolucion
;;                      sobre K1 y K2, con los literales repetidos
;;                      eliminados
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun resolve-on (lambda K1 K2)
 ;Si K1 o K2 son NIL, no hay nada que resolver.
 (if (or (null K1)
         (null K2))
     nil
  ;Siempre que K1 o K2 tengan el literal lambda
  ;en forma positiva o negativa:
  (when (resolvable-clauses-p lambda K1 K2)
   ;Obtiene las cláusulas resueltas.
   (let ((resolved-K1 (get-resolved-clause lambda K1))
         (resolved-K2 (get-resolved-clause lambda K2)))
    ;Si al resolver K1 y K2 ambas devuelven NIL,
    ;crea una lista de tipo (NIL).
    ;Sino, devolverá la unión entre las cláusulas resueltas.
    (if (and (null resolved-K1) (null resolved-K2))
        (list NIL)
     (list (union resolved-K1 resolved-K2)))))))
```

Para calcular el resolvente entre dos cláusulas $res_\lambda(K1, K2)$, la función `resolve-on (lambda K1 K2)` utiliza dos funciones auxiliares.

1. **`resolvable-clauses-p (lambda K1 K2).`** Comprueba si K1 y K2 se pueden resolver. Para ello, K1 debe contener un literal positivo λ y K2 un literal negativo (∼ λ), o viceversa.

2. **`get-resolved-clause (lambda K).`** Devuelve la cláusula K resuelta sobre λ.

   Si K contiene al literal positivo λ, se eliminarán todas sus ocurrencias de K. Por otro lado, si K contiene al literal negativo (∼ λ), también se eliminarán todas sus ocurrencias de K.

La función `resolve-on (lambda K1 K2)` comprueba primero si K1 o K2 son NIL. En ese caso, no hay nada que resolver.

Si K1 y K2 son distintas de NIL y además son resolubles, la función obtiene las cláusulas resueltas. Si en el proceso de resolución de K1 y K2 la función `get-resolved-clause (lambda K)` devuelve NIL (porque K1 y K2 tenían sólo un literal λ), la función devolverá (NIL).

En caso contrario, devolverá o bien $\{K1 - \{\lambda\}\} \cup \{K2 - \{\neg\lambda\}\}$, o bien $\{K1 - \{\neg\lambda\}\} \cup \{K2 - \{\lambda\}\}$; es decir, la unión de las cláusulas resueltas sobre λ.

Apartado 4.4.5

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.4
;; FUNCIONES AUXILIARES
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; resolvable-clauses-p (lambda K1 K2)
;;;
;;; Comprueba si dos cláusulas K1 y K2 se pueden resolver.
;;;
;;; INPUT: lambda: literal de la comprobación.
;;;                         K1: cláusula simplificada 1 de la comprobación.
;;;                         K2: cláusula simplificada 2 de la comprobación.
;;; OUTPUT: t si K1 y K2 se pueden resolver, NIL si no.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun resolvable-clauses-p (lambda K1 K2)
 ;O K1 tiene el literal lambda positivo y K2
 ;tiene el literal lambda negativo.
 (or (and (contains-positive-literal lambda K1)
        (contains-negative-literal lambda K2))
    ;O viceversa.
    (and (contains-positive-literal lambda K2)
        (contains-negative-literal lambda K1))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-resolved-clause (lambda K)
;;;
;;; Devuelve una cláusula K resuelta sobre lambda.
;;;
;;; INPUT: lambda: literal sobre el que se va a
;;;       resolver la cláusula.
;;;                         K: cláusula que se va a resolver.
;;; OUTPUT: Cláusula resuelta o NIL si K no contiene a lambda.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun get-resolved-clause (lambda K)
 (cond
  ;CASO 1: K contiene al literal positivo lambda.
  ;Todas las ocurrencias de lambda se eliminan de K.
  ((contains-positive-literal lambda K)
   (remove lambda K :test #'equal))
  ;CASO 2: K contiene al literal negativo (~ lambda).
  ;Todas las ocurrencias de (~ lambda) se eliminan de K.
  ((contains-negative-literal lambda K)
   (remove (list +not+ lambda) K :test #'equal))
  (t NIL))) ;No debería llegar a este caso nunca.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.4
;; resolvente de dos clausulas
```

```
;;
;; RECIBE   : lambda     - literal positivo
;;          K1, K2     - clausulas simplificadas
;; EVALUA A : res_lambda(K1,K2)
;;                    - lista que contiene la
;;                      clausula que resulta de aplicar resolucion
;;                      sobre K1 y K2, con los literales repetidos
;;                      eliminados
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun resolve-on (lambda K1 K2)
 ;Si K1 o K2 son NIL, no hay nada que resolver.
 (if (or (null K1)
         (null K2))
     nil
  ;Siempre que K1 o K2 tengan el literal lambda
  ;en forma positiva o negativa:
  (when (resolvable-clauses-p lambda K1 K2)
   ;Obtiene las cláusulas resueltas.
   (let ((resolved-K1 (get-resolved-clause lambda K1))
         (resolved-K2 (get-resolved-clause lambda K2)))
    ;Si al resolver K1 y K2 ambas devuelven NIL,
    ;crea una lista de tipo (NIL).
    ;Sino, devolverá la unión entre las cláusulas resueltas.
    (if (and (null resolved-K1) (null resolved-K2))
        (list NIL)
     (list (union resolved-K1 resolved-K2)))))))
```

Para calcular el conjunto de cláusulas $RES_\lambda(\alpha)$ para una FNC α, la función `build-RES (lambda cnf)` utiliza dos funciones auxiliares.

1. **`get-list-of-resolved-clauses (lambda K clause-set).`** Devuelve una lista de resolventes $res_\lambda(K, Ki)$, siendo K la cláusula pasada como argumento y Ki cada una de las cláusulas del conjunto clause-set. Todas las resoluciones son sobre λ.

   Para ello, resuelve K y la primera cláusula del conjunto clause-set s sobre λ. Si obtiene la cláusula vacía (NIL), devuelve dicha cláusula (no hay más cláusulas que resolver).

   Por el contrario, si la cláusula resuelta no es (NIL), construye una lista con esta cláusula y el resto de resoluciones entre K y cada cláusula de clause-set. Todo ello se realiza de forma recursiva.

2. **`get-all-resolved-clauses (lambda positive-clauses negative-clauses).`** Devuelve una lista de resolventes entre cada una de las cláusulas del conjunto $(\alpha_\lambda^{(+)})$ y las cláusulas del conjunto $(\alpha_\lambda^{(-)})$.

   Para ello, obtiene de forma recursiva la lista de cláusulas resueltas entre cada cláusula del conjunto lambda-positivo y las cláusulas del conjunto lambda-negativo.

Una vez obtenidos los distintos conjuntos de cláusulas resueltas por cada cláusula del conjunto $(\alpha_\lambda^{(+)})$, realiza una unión sobre ellos y elimina las posibles cláusulas repetidas.

Apoyándose en estas funciones, `build-RES (lambda cnf)` extrae los conjuntos de cláusulas lambda-positivos, lambda-neutros y lambda-negativos, y realiza la unión entre las cláusulas del conjunto $(\alpha_\lambda^{(0)})$ y las cláusulas resueltas entre $(\alpha_\lambda^{(+)})$ y $(\alpha_\lambda^{(-)})$ obtenidas a partir de `get-all-resolved-clauses (lambda positive-clauses negative-clauses).`

Apartado 4.5

Código

```
;;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.5
;; Comprueba si una FNC es SAT calculando RES para todos los
;; atomos en la FNC
;;
;; RECIBE   : cnf - FBF en FNC simplificada
;; EVALUA A :       T  si cnf es SAT
;;             NIL  si cnf es UNSAT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun  RES-SAT-p (cnf)
  (if (null cnf)
     ;Registramos el caso de la tautología.
     T
   ;Pasamos a la parte recursiva de la función
   (RES-SAT-p-rec cnf (make-literal-list cnf))))


;Implementación recursiva de la función, resuelve la cnf mediante cada uno de
;los literales que se encuentran en la misma, contenidos en lst
(defun RES-SAT-p-rec(cnf lst)
  (if (null lst)
     ;Si terminamos la lista...
     (if (null cnf)
        ;...Y cnf es null, devolvemos T, es SAT
        T
       ;...Y cnf no es null, devolvemos nil, es UNSAT
       nil)
   ;Comprueba que no haya clausulas vacias
   (let ((res (build-RES (first lst) cnf)))
      (if (or (check-4-empty-clause cnf)
              (check-4-empty-clause res))
         ;Si hay clausulas vacias, devuelve directamente nil
         nil
        ;Si no, resuelve frente al siguiente literal
        (RES-SAT-p-rec
         (simplify-cnf (build-RES (first lst) cnf))
         (rest lst))))))

;Genera la lista de literales de una cnf, todos en
;estado positivo
(defun make-literal-list (cnf)
  (eliminate-repeated-literals
        (make-all-positive (reduce #'union cnf))))
```

```
;Función que hace todos los átomos de una lista de ellos
;positivos
(defun make-all-positive (lst)
  (if (null lst)
      nil
    (cons
     (make-positive (first lst))
     (make-all-positive (rest lst)))))

;Función que busca, que devuelve la versión positiva
;del átomo dado atom
(defun make-positive (atom)
  (if (positive-literal-p atom)
      atom
    (second atom)))

;Función que busca, en cnf, clausulas vacías, devolviendo
;T si la encuentra, nil en caso contrario.
(defun check-4-empty-clause (cnf)
  (if (null cnf)
      nil
    (if (null (first cnf))
        T
      (or
       (check-4-empty-clause (rest cnf))))))
```

Para discernir si una cnf es SAT o UNSAT nos hemos enfrentado a un único problema, que es es de formar una lista con todos los literales (en estado positivo) presentes en una función.

Esto se ha llevado a cabo mediante la función auxiliar **make-literal-list**, a la cual, al introducir una cnf, primero reduce mediante la función *union*, lo cual te devuelve una lista con todos los literales positivos y negativos presentes en la lista. Seguidamente, aplicamos la función auxiliar **happiness,** que, a su vez, mediante la aplicación recursiva de la función **make-positive**, devuelve la lista con todos sus literales en estado positivo. Por último, mediante la función previamente codificada de **eliminate-repeated-literals**, la lista se queda con una única aparición de cada uno de sus clausulas como una clausula positiva.

Una vez llegados a este punto, simplemente hubo que reflejar los casos particulares de una cnf sin clausulas (en cuyo caso devuelve directamente *T*, dado que la cnf sería SAT) y el caso de que hubiera alguna clausula nula (comprobado mediante la función auxiliar **check-4-empty-clause** y devolviendo *nil* en dicho caso, dado que la cnf sería UNSAT) y aplicar de manera recursiva la función de resolución con cada una de las clausulas previamente obtenidas sobre la cnf, devolviendo *T* si se resuelve (sólo queda nil en la cnf) o *nil* en caso contrario.

Apartado 4.6

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.6:
;; Resolucion basada en RES-SAT-p
;;
```

```
;; RECIBE   : wff - FBF en formato infijo
;;          w   - FBF en formato infijo
;;
;; EVALUA A : T si w es consecuencia logica de wff
;;          NIL en caso de que no sea consecuencia logica.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun logical-consequence-RES-SAT-p (wff w)
  (if (null
       ;Comprobamos si es satisfactible o no...
       (RES-SAT-p
        ;La union...
        (union
         ;De la forma FNC de wff
         (wff-infix-to-cnf wff)
         ;...Y la forma FNC de ~w
         (wff-infix-to-cnf
          (list +not+ w)))))
      ;Si no lo es, w es consecuencia lógica de wff, devolvemos T
      T
    ;Si lo es, w NO es consecuencia lógica de wff, devolvemos nil
    nil))
```

Este último apartado del ejercicio 4 ha sido implementado siguiendo la fórmula propuesta en el guión de la práctica, pasando tando *wff* y ¬*w* (este último generado a partir de listar *+not+* y *w)* a FNC mediante la previamente implementada **wwf-infix-to-cnf** , uniéndolas mediante *union* y resolviendo mediante **RES-SAT-p**, realizada en el apartado enterior.

Si esto resulta *nil*, quiere decir que *w* es consecuencia lógica de *wff*, y por tanto, la función devolverá *T*. En el caso contrario, devolverá *nil*, dado que habremos discernido que no es consecuencia lógica.

Cabe destacar como reflexión que, aunque este apartado, en si, es terriblemente simple en comparación con el resto de apartados de este ejercicio, ha sacado a luz una serie de errores que habíamos arrastrado a lo largo de los apartados previos que no habían salido a luz en los casos de prueba propuesto, lo que ha derivado en que hayamos tenido que dedicar una mayor cantidad de tiempo de la que cabría esperar, pero ha mejorado en buena medida la calidad de nuestro código.

5. Búsqueda en anchura

Apartado 5.1

Se han resuelto el caso especial de un caso lineal, el grafo propuesto en el planteamiento del ejercicio y el grafo del ejercicio 5.7 como segundo grafo simple.

Para todos ello se han cogido un par de nodos comienzo y destino más alejados posibles en cada uno de los grafos.

En el caso del grafo lineal, se ha resuelto el caso iterando cada uno de los nodos del de comienzo al final.

En el caso del grafo del planteamiento del ejercicio, asumiendo que los nodos están 'ordenados' en base a la letra, y que, por tanto, tienen prioridad durante la ejecución, la búsqueda del camino del nodo C al F se extiende, por una iteración, al camino [CEB], pero es descartado en la siguiente iteración, que genera el camino [CEF].

En el caso del grafo del ejercicio 5.7, las iteraciones del algoritmo para hallar el camino de C a F generarían, en cada iteración, los caminos [CA], [CG], [CAB], [CAD], [CAE], [CGD], [CGE], [CGH], [CABD], [CABE], [CABF] y terminaría, habiendo encontrado el camino [CABF] que llega al nodo final.

Apartado 5.2

```
BFS(grafo, camino_recorrido, nodo_actual, nodo_final)
if nodo_actual == nodo final:
        return camino_recorrido + nodo actual;
else :
adyacentes = get_nodos_adyacentes(grafo, nodo_actual);
for each nuevo_nodo in adyacentes:
        if no_existe(nuevo_nodo, camino_recorrido)
                BFS(grafo, camino_recorrido+nodo_actual, nuevo nodo, nodo final)
```

Apartado 5.3

Nada que reportar.

Apartado 5.4

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Breadth-first-search in graphs
;;;
;;;
;;; INPUT:        end: nodo destino de nuestra búsqueda
```

```
;;;                     queue: cola con los caminos de búsqueda
;;;                     net: lista representativa de nuestra red de nodos
;;; OUTPUT:             lista con el camino más corto al nodo destino
(defun bfs (end queue net)
  (if (null queue)
      ;;Establecemos '() como 'caso base' que, en realidad, sirve para reportar
      ;;que la queue se ha quedado vacía y no hemos encontrado el camino
      '()
    ;Cogemos el primer camino de la cola
    (let* ((path (first queue))
           ;Reconocemos el primer elemento del camino como el nodo
           (node (first path)))
      (if (eql node end)
          ;Si hemos contrado el nodo destino, devolvemos el la inversa del camino evaluado actual
          (reverse path)
        ;Si no, metemos todos los caminos a los que se puede llegar desde el nodo identificado en la cola
        (bfs end
             (append (rest queue)
                     (new-paths path node net))
             net)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; new-paths
;;
;; INPUT:               path: camino actual
;;                      node: nodo actual
;;                      net: lista representativa de nuestra red de nodos
;; OUTPUT:              lista de nodos a los que se puede llegar desde el nodo actual
(defun new-paths (path node net)
  (mapcar #'(lambda(n)
              (cons n path))
    (rest (assoc node net))))           ;; Genera caminos nuevos juntando el camino previo a todos aquellos
                                        ;; con todos aquellos nodos accesibles desde el nodo actual
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Los comentarios han sido añadidos al código del ejercicio, presente en el fichero de código de la práctica.

Resumiendo, el código funciona mediante una cola en la que guarda cada camino realizado, siendo el primer nodo de estos el último nodo explorado del camino, y del cual se valorarán nuevos caminos, añadiendo a la cola ese camino con, ahora, el primer elemento siendo un nuevo nodo alcanzable desde el último nodo previo, una vez por cada elemento descubrible, a la cola.

Cabe añadir que, dada esta disposición de nodos en los caminos, antes de devolverlo se aplica un *reserse* sobre el camino para que vaya de nodo inicial a nodo objetivo.

Apartado 5.5

Código

```
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Shortest-path through Breadth-first-search in graphs
;;;
;;; Encuentra el menor camino dado que introduce el primer nodo
;;; en el grafo y ejecuta una búsqueda de anchura hasta el nodo
;;; destino, la cual, por defecto, encontrará el camino uno de
```

```
;;; (o el) caminos con menor longitud.
;;;
;;;
;;; INPUT:          start: nodo origen de nuestra búsqueda
;;;                 end: nodo destino de nuestra búsqueda
;;;                 net: lista representativa de nuestra red de nodos
;;; OUTPUT:         lista con el camino más corto al nodo destino
(defun shortest-path (start end net)
  (bfs end (list (list start)) net))
```

La función ha sido añadida al fichero de código y comentada en consecuencia.

Básicamente encuentra el mejor camino dado que, al comenzar el algoritmo, únicamente metes en la cola el nodo del comienzo y, dada la propia naturaleza del algoritmo de búsqueda por anchura, su ejecución siempre devolverá uno de los caminos más cortos (o el más corto si no hay otro similar) de los nodos que introduzcas en su primera ejecución hasta el nodo final.

Apartado 5.6

```
(shortest-path ’a ’f ’((a d) (b d f) (c e) (d f) (e b f) (f)))
```

1. bfs(f ((a)) ((a d) (b d f) (c e) (d f) (e b f) (f))
2. bfs(f ((d a)) ((a d) (b d f) (c e) (d f) (e b f) (f))
3. bfs(f ((f d a)) ((a d) (b d f) (c e) (d f) (e b f) (f))
4. bfs(f ((f d a)) ((a d) (b d f) (c e) (d f) (e b f) (f))
5. (a d f)

Apartado 5.7

Código

(shortest-path 'f 'c '((a b c d e) (b a d e f) (c a g) (d a b g h) (e a b g h) (f b h) (g c d e h) (h d e f g)))

La llamada que hay que hacer incluye la correcta representación del grafo como una lista de nodos y sus caminos y la señalización del nodo inicial y el final.

```
(shortest-path 'f 'c '((a b c d e) (b a d e f) (c a g) (d a b g h) (e a b g h) (f b
h) (g c d e h) (h d e f g)))
```

Apartado 5.8

Código

```
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; new-paths loop safe
;;
;; INPUT:          path: camino actual
;;                 node: nodo actual
;;                 net: lista representativa de nuestra red de nodos
;; OUTPUT:         lista de nodos a los que se puede llegar desde el nodo actual
(defun new-paths-ls (path node net)
  (mapcar #'(lambda(n)
```

```
        (if (null (member n path))
            (cons n path)
          nil))
   (rest (assoc node net))))          ;; Genera caminos nuevos juntando el camino previo a todos aquellos
                                      ;; con todos aquellos nodos accesibles desde el nodo actual

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Loop safe Breadth-first-search in graphs
;;;
;;;
;;; INPUT:          end: nodo destino de nuestra búsqueda
;;;                 queue: cola con los caminos de búsqueda
;;;                 net: lista representativa de nuestra red de nodos
;;; OUTPUT:         lista con el camino más corto al nodo destino
(defun bfs-ls (end queue net)
  (if (or (null queue))
      ;;Establecemos '() como 'caso base' que, en realidad, sirve para reportar
      ;;que la queue se ha quedado vacía y no hemos encontrado el camino
      '()
    ;Cogemos el primer camino de la cola
    (let* ((path (first queue))
           ;Reconocemos el primer elemento del camino como el nodo
           (node (first path)))
      (if (eql node end)
          ;Si hemos contrado el nodo destino, devolvemos el la inversa del camino evaluado actual
          (reverse path)
        ;Si no, metemos todos los caminos a los que se puede llegar desde el nodo identificado en la cola
        (bfs-ls end
                (append (rest queue)
                        (new-paths-ls path node net))
                net)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Shortest-path through Loop safe Breadth-first-search in graphs
;;;
;;; Encuentra el menor camino dado que introduce el primer nodo
;;; en el grafo y ejecuta una búsqueda de anchura hasta el nodo
;;; destino, la cual, por defecto, encontrará el camino uno de
;;; (o el) caminos con menor longitud.
;;;
;;;
;;; INPUT:          start: nodo origen de nuestra búsqueda
;;;                 end: nodo destino de nuestra búsqueda
;;;                 net: lista representativa de nuestra red de nodos
;;; OUTPUT:         lista con el camino más corto al nodo destino o null si no hay camino
(defun shortest-path-ls (start end net)
  (bfs-ls end (list (list start)) net))

(shortest-path-ls 'a 'b '((a d) (b a d) (d b) (c a))) ;; -> (A D B)
(shortest-path-ls 'a 'c '((a d) (b a d) (d b) (c a))) ;; -> NIL
```

Añadiendo al bucle más interno de la ejecución del algoritmo (**new-paths**) un simple control en la función lambda, mediante el cual comprobamos si un nodo ya está presente en el camino antes de introducirlo como un nuevo camino a explorar en la cola.

Así, conseguimos, efectivamente, que en los caminos de nuestro algoritmo no se genere ningún bucle y, de esta manera, no sólo estaremos evitando el problema principal de los bucles infinitos en el caso de ser el nodo final inalcanzable, si no que estaremos ahorrando trabajo en la resolución de grafos de nodos entre los que hay caminos y algún posible circuito entre medias.

Como casos de prueba, se han utilizado las siguientes ejecuciones de la función, cada una dando, como se puede comprobar en la ejecución del código, el resultado esperado indicado en el comentario.

```
(shortest-path-ls 'a 'b '((a d) (b a d) (d b) (c a))) ;; -> (A D B)

(shortest-path-ls 'a 'c '((a d) (b a d) (d b) (c a))) ;; -> NIL
```

Como última aclaración, el *ls* añadido al añadido al nombre de todas las funciones de esta sobreimplementación del algoritmo significa "*Loop Safe*".